第3章 トラブルの芽と工夫の勘所を知る

事例で学ぶ 組み込みシステム開発

國方則和，木下秀昭，山崎辰雄

ここでは，さまざまなトラブルや問題解決の事例を読み解くことにより，実践的な組み込みシステム開発のノウハウを解説する．（編集部）

本稿で解説する事例一覧

1. 組み合わせテストでDSPの演算処理がおかしくなる（リセット時のタイミング設計）
2. ピックアップが止まらない（機構制御）
3. 割り込み要求が消えてしまう（割り込み制御レジスタのリード・モディファイ・ライト）
4. 優先したい割り込みが保留されてしまう（割り込み保留命令の予期せぬ影響）
5. デコーダのバッファを小さくしたい（データ読み出しと書き込みの工夫）
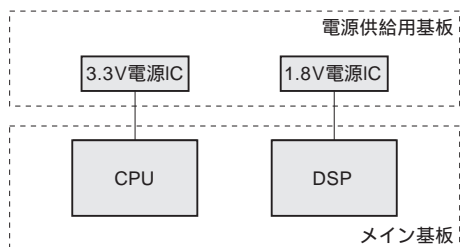
図1-1　システムの電源構成
今回のシステムでは，CPUとDSPの電源は別系統となっていた．同系統だとタイム・ラグは起きにくい．

事例1　組み合わせテストでDSPの演算処理がおかしくなる（リセット時のタイミング設計）

● 状況

ディジタル・オーディオ・レコーダの開発プロジェクトで，メイン基板と電源供給用の基板を組み合わせてテストを進めていました．メイン基板には，CPUとオーディオ・データ用のDSP（信号処理プロセッサ）を実装していました．ところがメイン基板の単体テストではすべての回路が正常に動作していたにもかかわらず，組み合わせテストではDSPの演算処理が正しく実行できていないことが分かりました．

● 原因究明

調査した結果，問題の原因はCPUとDSPのリセット・タイミングの差にあることが分かりました．システムの起動時の様子を詳しく見ると，CPU用の3.3V系電源よりDSP用の1.8V系電源の立ち上がりが200msほど遅れていることが観測できました（**図1-1**）．また，電源の立ち上がりが遅れることによって，DSPのリセット・タイミングがメイン基板の単体テストの時よりも遅れていることが分かりました．

デバッガを使ってDSPの設定レジスタを順番に読み出してみると，DSPとオーディオ・データの入出力用デバイスとのシリアル通信に関する設定がおかしくなっていました．

このDSPは起動後にシリアル通信の設定を切り替えるこ

KeyWord　リセット・タイミング，アクチュエータ，サーボ，割り込み要求フラグ，リード・モディファイ・ライト，ポーリング，割り込み保留命令，無線通信，エラー訂正，エンコード，デコード，バッファ

とができるため，CPUが転送データ長に関するDSPの設定レジスタを書き換える処理をプログラムしていました．ところが，プログラムでは特にタイミングを意識せずにシリアル通信の設定を切り替える処理を実装しており，DSPの起動が完了する前にシリアル通信の設定処理を実行していることが分かりました(**図1-2**)．

● 対応：仕様を再確認して特定の信号を利用

対応策を検討する際に，DSPの仕様書を確認したところ，起動が完了して処理が可能になったことを示す信号の出力があることが分かりました．そこで，この信号線をCPUの空ポートへ入力するように回路を変更しました．そして，この信号線の状態を読み出してDSPが正常に起動したことが確認できてから，シリアル通信の設定を切り替える処理を実行するようにプログラムを変更しました．

● 再発防止策：タイミング設計は綿密に

システム起動時の電源の立ち上がり方は，各回路の負荷の状態によって変動することがあります．このためリセット解除後の初期化処理などについては，ハードウェアも含めた綿密なタイミング設計が欠かせません．ソフトウェア技術者はハードウェア技術者と連携して，リセットが解除されるタイミングの変動範囲を明らかにした上で，変動を吸収するための仕組みをプログラムにあらかじめ組み込んでおくことが肝要です．

さらに電源やリセット回路の不具合発生に対して，タイムアウト処理やリカバリ処理をプログラムに組み込んでおくことによって，2次不具合の発生を未然に防ぐことができます．また不具合の発生を知らせるアラート機能を用意することで，迅速に不具合原因を特定できるようになり，保守性が向上します．

開発中の単体テストでは安定化電源装置などを使って配線基板に電源を供給することが多いため，今回の例のような問題に気が付かず，実際の製品で使用する基板と組み合わせたテストで初めて問題が発覚する場合は多いです．さらに，リセットのようにシステムが定常状態へ移行する前の過渡状態における不具合は，ICE(in-circuit emulator)を使ったデバッグでは発見するのが難しいため，机上でのタイミング設計と事前の対応策の組み込みが特に重要になります．

くにかた・のりかず
ティアック(株)

図1-2　システムのリセット解除と処理のタイミング図

電源の立ち上がりに遅延があるため，CPUとDSPのリセット解除タイミングがずれている．また，DSPのリセット解除前にシリアル通信の設定処理が実行されていたため，正しく設定できていなかった．

事例2　ピックアップが止まらない(機構制御)

● 状況

CDプレーヤの開発プロジェクトで，量産試作サンプルを使った負荷試験を実施していました．ところが，データ読み出しのためにCDドライブの光ピックアップをある程度以上移動させた後の，ディスクからのデータ読み出しが始まらないとの報告がありました．

● 原因究明

開発チームは試作サンプルを使って再現試験を行いましたが，指摘された現象は発生しません．開発側の評価環境と負荷試験環境では動作条件は同じです．唯一の違いは，光ピックアップや駆動する機構，サーボ回路などのサンプル個体だけです．

現象が発生しているCDドライブを取り寄せて再現実験を実施したところ，光ピックアップを高速移動した後に，光ピックアップが小刻みに振動している様子が確認できました．デバッガを使用してプログラムの実行状況をトレースしたところ，ディスク上の移動目標となる位置付近で，光ピックアップが反復移動を繰り返していることが分かりました．

CDドライブは，ディスク上のデータが書き込まれている位置に光ピックアップを移動し，レーザ光を照射して

ディスク上の位置情報を走査しながら，データを読み出していきます(**図2-1**)．光ピックアップにはレーザ光を照射するための対物レンズがあり，ある程度の範囲で動けるように取り付けられています(2軸アクチュエータで動かしている)．そしてディスク上のデータを読み出せるように対物レンズの位置を制御します．

光ピックアップのサーボ制御機構(与えられた制御値で作動する仕組み)としては，レーザ光の焦点を合わせるフォーカス・サーボと，レーザ光の走査位置を合わせるトラッキング・サーボがあります．フォーカス・サーボはアクチュエータの可動範囲内で制御が可能ですが，トラッキング・サーボはディスクの半径方向のすべての範囲をカバーするために，アクチュエータの位置制御だけでは十分ではありません．そのためアクチュエータの位置制御と光ピックアップ本体の位置制御を組み合わせて制御しています．
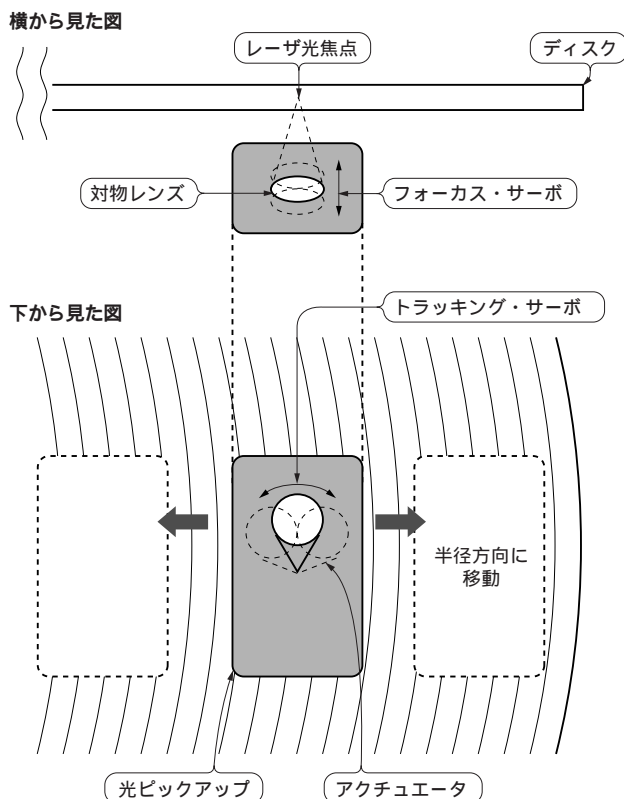
図2-1　CDドライブの光ピックアップ・サーボ

CDドライブは，ディスク上のデータを読み出すために，レーザ光を照射する対物レンズの位置を制御する．対物レンズの位置は，光ピックアップの位置制御と，アクチュエータの半径方向のトラッキング・サーボ，焦点方向のフォーカス・サーボを組み合わせて制御する．なお実際には，ディスクの回転速度も制御する．

例えば，レーザ光の照射位置が目的の場所から離れている場合，アクチュエータの可動範囲内であればアクチュエータだけを制御して位置を制御できます．ただしアクチュエータはその可動範囲の中点付近にいることが望ましいため，移動範囲がアクチュエータの可動範囲のあるしきい値を超える場合には，光ピックアップ本体を移動します．

ところが，光ピックアップ本体を動かすと，アクチュエータは慣性のため引きずられるように動き始めます．また光ピックアップ本体が停止すると，反対に移動方向へ押し出されます．つまり光ピックアップ本体の動きによって，アクチュエータは可動範囲内で右往左往する訳です．開発チームでは，これを「往復ビンタ現象」と呼んでいました(**図2-2**)．また，光ピックアップ本体の移動加速度が大きければ大きいほど，アクチュエータの振られる範囲は大きくなります．

ここで問題が発生します．例えば，半径方向で外側に光ピックアップ本体を移動してアクチュエータで位置を微調整する場合，光ピックアップ本体は目標位置の中心付近にいるにもかかわらず，アクチュエータは中心から大きく離れた位置にいる可能性があります．このとき単純な制御ロジックで対応すると，光ピックアップ本体を反対方向の内側に引き戻すように制御してしまいます．ところが引き戻した際にもアクチュエータの位置が振られていると，再び外側方向に引き戻すように制御してしまいます．このように，ある状態で均衡してしまうと，永久にこの動作を繰り返す結果となります．今回の事例においても，まさにこの
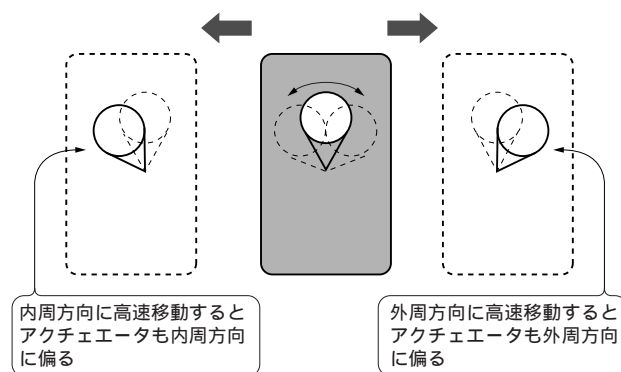
図2-2　アクチュエータの「往復ビンタ現象」

光ピックアップを高速に移動すると，内部のアクチュエータが慣性によって右往左往する．

現象が発生していました．

● 対応：制御方式を一工夫する

そこで制御方式を変更して，同じ範囲の移動を繰り返す場合は光ピックアップ本体の移動距離を徐々に短くすることで移動加速度を落とし，現象が収束していくようにプログラムを修正しました．なお，この制御方式は特許登録されています（特許番号は第2947095号）．

● 再発防止策：ハードウェアの振る舞いを理解する

いくつかの制御系を持つシステムの場合，互いが干渉して複雑な振る舞いをすることがあります．可動部が多くなればなるほど，さらに複雑な干渉が発生します．ハードウェア技術者は，できるだけ干渉が発生しないように，あるいは干渉が動作に影響しないようにあらゆる手を尽くして設計しますが，それでも防ぎきれない状況は発生します．

そのような場合，ソフトウェア技術者にも対応に参画することが求められます．その際，ソフトウェア技術者がハードウェアの動きを理解できていないと，まともな対応はできません．ソフトウェア技術者もハードウェアの動作原理を理解し，説明できる程度の知識を身に付けておくとよいでしょう．

また，「ばらつき」という概念も知っておく必要があります．ソフトウェアはマイコンの中の論理的な条件だけで動作しますが，システムの振る舞いはマイコンの外側にある自然界のさまざまな制約条件によって変化します．これを考慮しなければ，組み込みシステムの制御はできません．

このようにソフトウェア技術者は，制御対象であるハードウェアに常に関心を持っている必要があります．また，不具合の発生を未然に防ぐためには，ハードウェア技術者と綿密に連携していくことが不可欠です．

くにかた・のりかず
ティアック（株）

<筆者プロフィール>

國方則和．1989年，ティアックに入社．音響機器のアナログ，ディジタル回路設計，基板レイアウト設計などを担当．1998年よりHDDレコーダのソフトウェア開発に従事，品質改善活動や人材育成活動にも取り組む．組込みソフトウェア管理者・技術者育成研究会（SESSAME）メンバ．

事例3　割り込み要求が消えてしまう（割り込み制御レジスタのリード・モディファイ・ライト）

● 状況

デバイス・ドライバとアプリケーション・ソフトウェアを組み合わせたソフトウェアを開発して検証を行っていたところ，デバイス・ドライバ側で使用している割り込み要求がまれに抜けてしまう，という現象が発見されました．

原因を調べると，どうやら割り込みハンドラ・ルーチン内の割り込み要求フラグをソフトウェアでクリア処理している個所が悪さをしているらしいことが分かりました．

リスト3-1はそのソース・プログラム例です．①で割り込み要求フラグを論理積演算でクリアしています．ただし，発生した不具合は，ここでクリアしているフラグ（IF0レジスタのビット0）に対応した割り込みが抜けているわけではありません．クリアするタイミングが悪いといった問題ではないようです．

● 原因究明

リスト3-1の①でアクセスしているIF0というレジスタは，CPU内蔵の各種周辺機能で使用する割り込み要求フラグを集めたレジスタです．そして，今回発生した現象である「割り込み抜け」の起こる割り込みのフラグは，このIF0に含まれていました．**リスト3-1**のC言語プログラムをコンパイルした結果を**図3-1(a)**に示します．①元の値を読み（リード），②論理積演算を行い（モディファイ），③レジスタに書き戻す（ライト）という三つのアセンブリ命令に展開されています．

通常はこれで問題ないのですが（**図3-1(b)**），IF0レジスタを読み出した直後に割り込みが発生した場合に，以下のような問題が起こってしまっていました（**図3-1(c)**）．

リスト3-1　割り込みハンドラ・ルーチンのプログラム例

```
void Vect01( void )    /* 割り込みハンドラ・ルーチン */
{
  IF0L &= 0xFE;        /* IF0レジスタのビット0をクリア */ …①

  (中略)

  return;
}
```
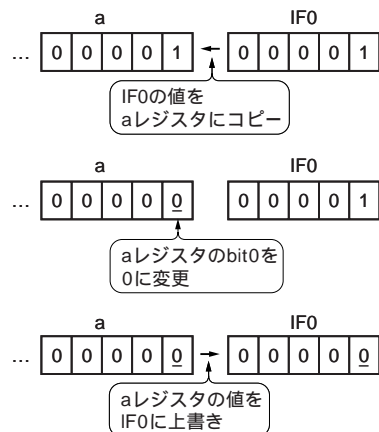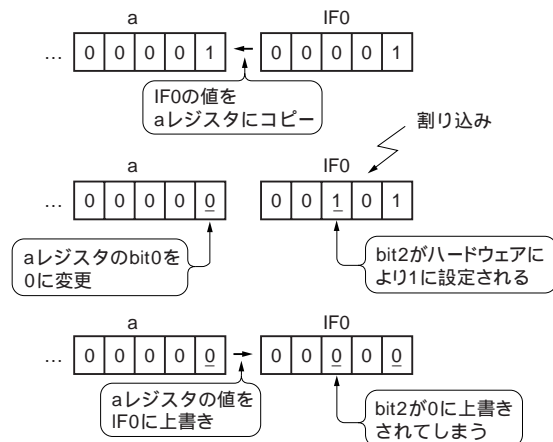
図3-1　リード・モディファイ・ライト

C言語では1行でも，実際（アセンブリ・コード）は3ステップで実行される．通常は（b）のように問題なく動作するが，（c）のような場合には問題となる．このような特定のタイミング依存の不具合は，解析や再現に手間取ることが多い．

- IF0レジスタに割り込み要求フラグが設定される
- IF0レジスタの値をaレジスタにコピーする
- IF0レジスタに新しい割り込み要求フラグが設定される
- aレジスタで割り込み要求フラグに対応したビットの値を変更する
- 先に設定された要求フラグをクリアするため，aレジスタの値をIF0レジスタに上書きする（新しい割り込み要求フラグは処理されないまま消されてしまう）

● 対応：CPUの仕様を再確認する

対応としては，リード・モディファイ・ライトをなくすことになるのですが，特にC言語で記述する場合，使用するCPUやコンパイラのマニュアルを参照して適切な対応をとる必要があります．今回発生したCPUのマニュアルとFAQを確認したところ，ビット操作命令を使用することで割り込み抜けが発生しないことが分かりました．また，そもそも今回のCPUでは割り込み要求フラグは割り込み受け付けと同時にハードウェアにてクリアされるのでソフトウェアでのクリアは必須ではなかったことが分かりました．

● 再発防止策：アセンブリ・コードまで想像する

このように，ハードウェアの動きを考慮せずにソフトウェアを書いてしまうと思わぬ落とし穴にはまることがあります．この手の問題は再現性が100％ではないことが多く，直接ソフトウェアだけを見ていても発見することは困難です．ハードウェアの動作を十分考慮した設計を心がけましょう．

きのした・ひであき
NECマイクロシステム（株）

事例4　優先したい割り込みが保留されてしまう（割り込み保留命令の予期せぬ影響）

● 状況

とあるCPU用のテスト・プログラムを作成していたときのことです．CPU内のタイマを利用してタイミング調整をするために，割り込み要求フラグをポーリングするルーチン（**リスト4-1**）を作成しました．しかし，動作確認したところ，ポーリング中にほかの割り込みが発生してもハンドラにジャンプせず，ポーリングが終了（この場合はタイマ割り込み発生）するまで保留されてしまう現象が発生しま

リスト4-1　タイマの割り込み要求フラグをポーリングするプログラム例

ハードウェア制御では，しばしばこのような処理が登場する．

```
do{              /*タイマ割り込みが発生するまでポーリング */
     ;           /*処理なし */
}while(IF0.0==0);/*IF0レジスタのビット0をチェック */
```

リスト4-2　リスト4-1のコンパイル結果
対象レジスタ・ビットをオペランドに持つ1命令の繰り返しになっている．

```
__LABEL__:
    BF  IF0.0, __LABEL__    …①
   #IF0の0bit目が‘0’ならば__LABEL__にジャンプ
```

リスト4-3　割り込み要求フラグの確認を関数にしたプログラム例
単一命令の繰り返しにならないように工夫する．

```
do
{
    Result = GetStatusTimer();  /* タイマ状態確認 */
} while (Result == BUSY )       /* タイマ待ち */
```

図4-1　IF0.0の割り込み要求フラグをポーリングした際の処理状況
リスト4-2の命令①がほかの割り込み要求の受け付けを保留してしまうため，IF0.0の割り込み要求が発生するまで，ほかの割り込みはすべて保留状態となってしまう．

した．

割り込み要求フラグがセットされていることはデバッガ画面で確認済みです．割り込みが受け付け可能な状態（enable interrupt）であることも確認できています．しかし，なぜか割り込みハンドラにジャンプしてくれません．

● 原因究明

ポーリングを行っている個所をよく確認すると問題があることが分かりました．CPUのマニュアルによると，命令の中には，実行中に割り込み要求が発生しても，命令の実行が終了するまでほかの要求の受け付けを保留するものがあると記載されています．その中に，割り込み制御レジスタにアクセスする命令が含まれていました．

今回の例の場合，IF0レジスタへのアクセスが割り込み保留の対象となります．さらに詳しく見るため，コンパイル後のアセンブリ命令を確認したものが**リスト4-2**です．割り込み保留となるIF0へのアクセス命令を連続して実行する内容となっています．これにより，ポーリング中は割り込みが保留されてしまっていたのでした（**図4-1**）．

● 対応：使用する命令を工夫する

このような場合は，単純なポーリングによるループとするのでなく，例えばフラグを確認する処理を関数にしてフラグを直接ポーリングしない，あるいはループの中に割り込み受け付け可能な命令を挿入しておくなどの対策が必要となります（**リスト4-3**）．

● 再発防止策：CPUの制限事項を確認しておく

使用するCPUのアーキテクチャによって，ソフトウェア技術者が予期しないような制限事項が存在することがあります．事前によくチェックして，対策をレビューしておきましょう．

きのした・ひであき
NECマイクロシステム（株）

事例5　デコーダのバッファを小さくしたい（データ読み出しと書き込みの工夫）

● 状況

無線通信機器の開発において，パケット多重データを復号化するデコーダ（復号器）を設計したときのことです．通常はバッファを二つ用意するのですが（ダブルバッファ），何とかしてバッファを小さくできないか，と要望されました．

● エラー訂正を考慮する

無線通信でデータを送受信する場合，送信機側では，送りたい情報を符号化し，変調して無線電波に乗せます．受信機側では，受けた電波を復調し，デコード（復号化）して送られた情報を取り出します．

例として，無線通信で“abcdefghijklmnopqrstuvwx”の24文字を送ることを考えてみましょう．無線なので，電波状況によってはデータの一部が欠落し，文字が化けてしまうことがあります．その対策として，送りたい情報以外にエラー検出訂正用のコード（訂正符号）を付けます[注5-1]．こ

注5-1：エラー訂正は通常ビット単位で行う．ここでは話を簡単にするために，ビットを英字に置き換えて表現している．
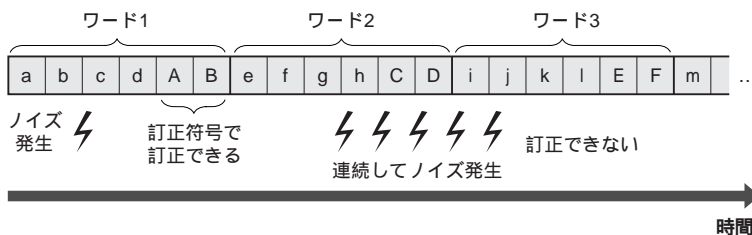
図5-1　ノイズ発生とエラー訂正
単発のノイズであれば，訂正符号で復元できる．しかし，無線通信のノイズは連続して起こることが多い．ワードを構成するビットが複数連続で破壊されると，訂正できなくなる．

図5-2　複数のワードをまとめてから並び替えて送信
複数のワードを束ねてから送ると，連続的なノイズに強くなる．6文字連続のエラーも，受信側で並べ直せば各ワードあたり1文字のエラーに相当するので，訂正できる．
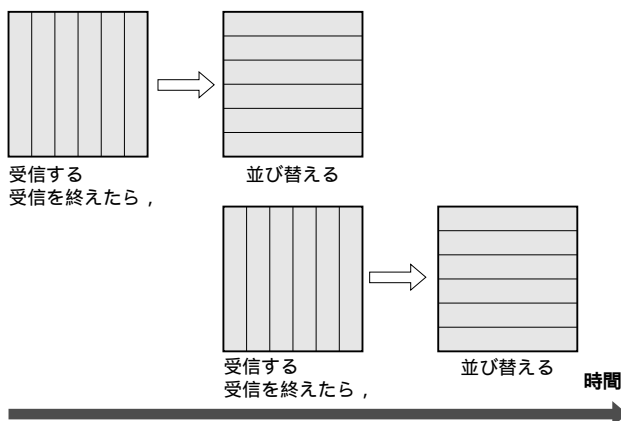
図5-3　デコード処理中にも受信するためのバッファが必要
処理速度の異なる物の間で情報を受け渡しするときには，バッファが必要となる．ここでは，受信するためのバッファと，受信した後に並び替えに使うバッファを用意している．
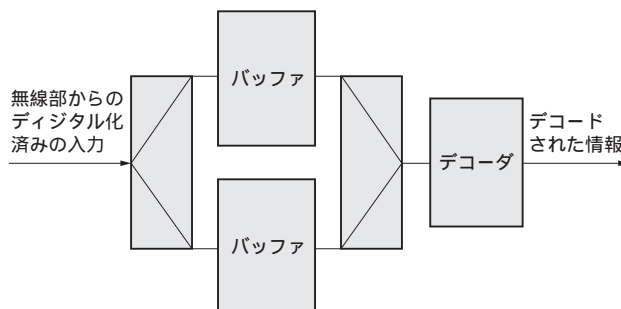
図5-4　通常の構成
受信中のデータを保持するためのバッファと，受信完了後のデコード処理中にデータを保持するバッファ，縦方向に記録されている6×6の表を横方向に読み出すためのデコーダが必要になる．

こでは4文字ごとに二つの訂正符号を付けることにしましょう．すると送信する文字列は次のようになります．

“ abcdABefghCDijklEFmnopGHqrstIJuvwxKL ”

この訂正符号を使うことで，1文字分の情報が壊れても訂正でき，2文字分なら誤りを検出できるものとします．これで満足してはいけません．無線で起きるノイズは，一度発生するとやや長時間続くものです(**図5-1**)．だからといって訂正符号をさらに付けても，今度は訂正符号の割合が増えるばかりで効率的ではなくなります．

● ノイズに強い形でデータを送る

そこで，次のような方法を考えました．6×6のマス目を用意して，そこにこれらの符号を六つずつ埋めていきます．まず横方向に埋めていきましょう．これを今度は縦方向から順に取り出します．すると取り出された符号は次のようになります(**図5-2**)．

“ aeimqubfjnrvcgkoswdhlptxACEGIKBDFHJL ”

このように並び替えて送ると，6文字連続のノイズを受けても訂正することができるようになります．これは，通信中の6文字連続のエラーが，受信側で元の順番に戻したときには訂正符号に対してそれぞれ1文字ずつのエラーになるからです[注5-2]．

バッファにいったん取り込んで，取り込み終えたらデコードに取りかかります(**図5-3**)．これを普通に実現すると，6×6の入力バッファが二つ必要になります(**図5-4**)．この「バッファが二つ必要」ということが，表が大きくなるにつれて目についてきます．これがもったいないのです．

● バッファからの読み出し・書き込みを工夫する

そこでアルゴリズムで一工夫してみました．バッファとデコーダを統合してしまうのです(**図5-5**)．するとバッファ

注5-2：この手法も大きな意味で，エンコード(符号化)・デコード(復号化)である．
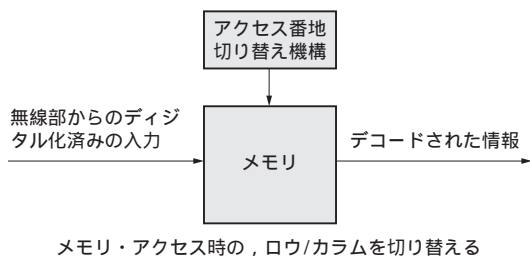
図5-5　工夫した構成
メモリに対して，アクセス番地を切り替えるだけで，バッファとデコーダの働きを兼務させる．
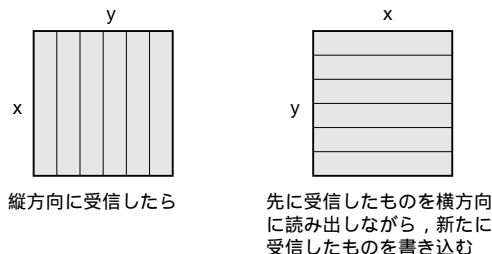
図5-6　メモリ領域(バッファ兼デコーダ)の使い方
メモリ・セルへの読み出し/書き込み方向を変えながら，読み出しと書き込みを並行して行うことで，デコードしながら受信データのバッファリングを行っている．

が一つ不要になります．

受信したとき，最初はバッファに縦方向に書き込みます．縦方向に6ワード分受信したら，次に受信するときには横方向に書き込みます．この書き込みの前に，バッファから横方向にデータを読み出すのです(**図5-6**)．さらに受信したときには，再び縦方向に書き込みます．この書き込みの前に，縦方向に読み出します．

このように読み出し方や書き込み方を工夫し，読み出しながら書き込めるようにしたことにより，バッファを半減できました．この方法は，エンコーダにも応用できます．

やまさき・たつお
ニュアンス コミュニケーションズ ジャパン(株)

コラム

天はハードの上にソフトを造らず，ハードの下にソフトを造らず

あなたは「ハード屋」さんですか？「ソフト屋」さんですか？

大まかに言って，機構設計を担当している技術者(いわゆる「メカ屋」さん)や電気回路を設計している技術者(いわゆる「エレキ屋」さん)は，「ハード屋」と呼ばれることが多いと思います．また，ハードウェア制御のプログラムを設計する技術者(いわゆる「ファーム屋」)やアプリケーション・ソフトウェアを設計する技術者は，「ソフト屋」と呼ばれることが多いですね．

さて，ハード屋とソフト屋という言葉は，どのように使い分けられるのでしょうか．単純な答えは，「ハードウェアを設計する人がハード屋，ソフトウェアを設計する人がソフト屋」なのでしょうが，筆者は，もっといろいろな見方があると思っています．

例えば，コストや納期などといった外的な制約がとても厳しい仕事を任されている人が「ハード屋」，比較的ゆるい制約下の仕事をしている人が「ソフト屋」という見方はどうでしょう？ きっと「それなら，みんなハード屋じゃないか！」と言われそうですが，制約というものをもっといろいろな切り口でとらえてみると面白いと思います．

例えば，汎用の部品を組み合わせて設計する人は「ハード屋」，自由な構想で部品から設計する人は「ソフト屋」，と考えてみたらどうでしょうか．あるいは，開発日程上のマイルストーンが生産工場を中心に設定されている場合に，(特殊な部品の発注や複雑な金型の作成スケジュールなどは融通が利きにくいため)生産スケジュールに影響を及ぼす仕事をしている人を「ハード屋」，生産が始まるまでに成果物を完成させる仕事をしている人を「ソフト屋」，と考えることもできると思いませんか．

賛否両論はあると思いますが，いずれにしてもここで大事なのは，どのような立場であれ「みんなで協力して，一つの物(製品)を創り出している」ということです．

物を“創り出す”ためには，多くの人がそれぞれの役割で責任を果たしていくことが大切です．ところが現実には，さまざまな制約やトラブルにより，思ったように責任を果たせなくなることもあります．そんなときに融通が利くのであれば，誰かが手助けをしてあげるというのは自然な姿です．ハード屋さんは，自分だけではどうにもならない状況の中で仕事をしているかもしれません．そこをソフト屋さんが補ってあげることで，ひとつの物が出来上がっていきます．

創り出した物で，人を喜ばせることができる，人を助けることができる，どんな物にもそういった価値があるはずです．ハード屋さんだろうが，ソフト屋さんだろうが，みんなで大きな責任を果たすことで価値のある物が出来上がるのなら，とても仕事が楽しくなるはずです．価値を持った物をどうやったら創り出せるのか，一緒になって考えていくことが，とても大切なことだと思います．

あなたは，ハード屋さんですか？ ソフト屋さんですか？ それとも「物作り屋さん」ですか？

くにかた・のりかず
ティアック(株)